Operator Implementation Wrap-Up
Query Optimization

Last time:

- Nested loop join algorithms:
  - TNLJ
  - PNLJ
  - BNLJ
  - INLJ
- Sort Merge Join
- Hash Join

General Join Conditions

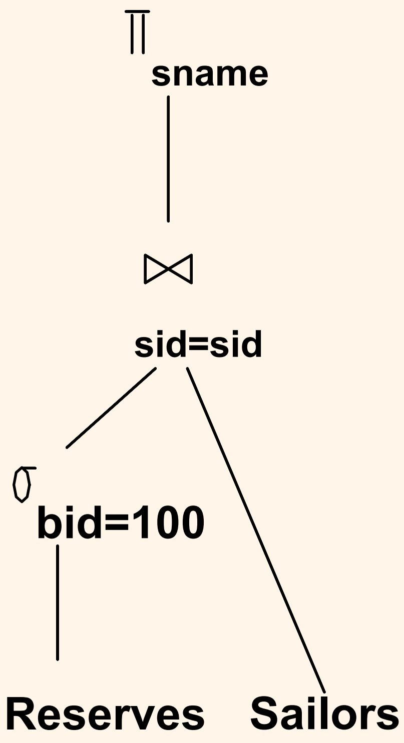
- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - Index NL works if we have an index on both sid and sname (together or separately)
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

General Join Conditions

- Inequality conditions (e.g., *R.rname* < *S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NLJ quite likely to be the best join method here.

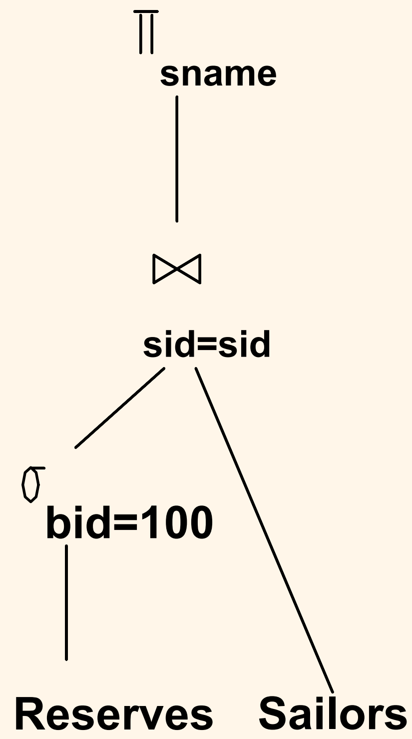
Blocking vs non blocking algorithms

- Suppose your join is not evaluated in isolation, but sits within a bigger RA tree
- Interesting to think about overall evaluation



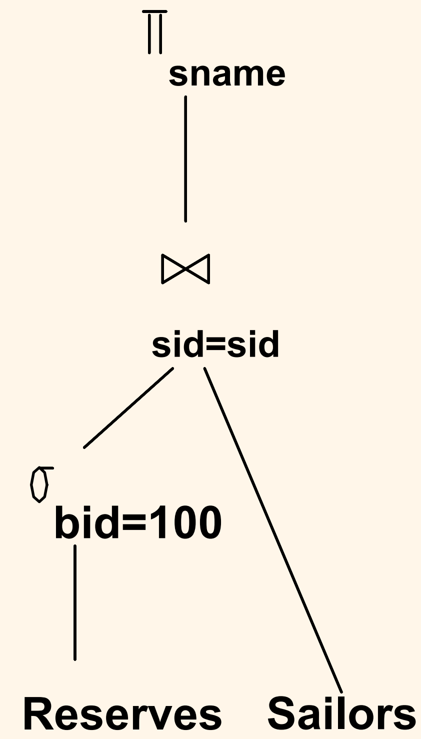
Blocking vs non blocking algorithms

- Selection produces R tuples one at a time
- Some join algorithms can get started right away, others can't



Blocking vs non blocking algorithms

- Hash join has to wait for all of the $\sigma(R)$ tuples - **blocking**
- Nested loops joins can get started right away – **non blocking**
- Sort-merge join?
- Note: not the same usage of "blocking" as in BNLJ!!
  - Here: blocking = needs to read at least one input completely before producing output



Other implementations

- The implementations you have seen are geared to specific requirements
  - Compute **entire** result **as fast as possible**
- Sometimes you may have a setting with different desiderata
  - Which may call for totally different implementations!

Case Study: Online Aggregation

- Setting: interactive exploration of large data sets to discover general trends
- Example:

```
SELECT P.zipcode , AVG(E.salary)
FROM EmploymentData E, PersonalData P
WHERE E.ssn = P.ssn
GROUP BY P.zipcode;
```

Case Study: Online Aggregation

- This query may take a lot of time to compute
- But we often don't need precise results
- Some estimate (with a confidence interval) is enough
- So, goal is to compute partial results fast
  - And give a running confidence interval so user can stop evaluation once satisfied
  - Means we want to sample from DB (definitely don't want sorted order!)

Joins for online aggregation

- Definitely do not want to have to read both relations before starting to output result
  - So the blocking algorithms (hash join and sort merge) are out

Joins for online aggregation

- Nested loops might work:
  - Pick *random* tuple from R (not trivial!)
  - Join it with all of S, update confidence interval
  - Get another random tuple and repeat until enough
  - Note would want smaller relation as inner because can update confidence intervals more often

Ripple joins

- Even better
- Avoid scanning all of the inner relation each time
- Basic idea:
  - Pick random tuple from R and random tuple from S
    - Join them
  - Pick two more random tuples, one from each
    - Join all 4 tuples together
  - Continue until confidence interval acceptable

Ripple Join: Square Two-Table Join

	R
S	X

Ripple Join: Square Two-Table Join

	R	
S	X	X
	X	X

Ripple Join: Square Two-Table Join

	R		
S	X	X	X
	X	X	X
	X	X	X

Ripple Join: Square Two-Table Join

	R			
S	X	X	X	X
	X	X	X	X
	X	X	X	X
	X	X	X	X

Pseudocode

```
for (max = 1 to infty)
        for (i = 1 to max – 1)
                if (match (R[i], S[max])
                        output tuple
        for (i=1 to max)
                if (match (R[max], S[i]))
                        output tuple
```

Ripple Joins

- Can have non-square aspect ratios if desired due to statistical properties of data
  - i.e. if want to sample one relation more often than the other
- This method of join evaluation allows confidence intervals to shrink quite fast
  - Theory and experimental results in research paper if you're interested (link in CMS)

Ripple Joins

- Extremely inefficient if we wanted to compute the whole join
- Even worse than tuple nested loops join
  - Because would need to keep track of the already-seen tuples from both relations (eventually won't fit in memory)
- Can use indexes, blocking or hashing to help
- But the main reason this works is that we almost always stop computation very early

Set Operations

- Intersection and cross-product special cases of join.
  - Intersection: equality on all fields is join condition
  - Cross product: no equality condition

Set Operations - Union

- Main challenge: eliminating duplicates
- Sorting based approach
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.

Set Operations - Union

- Hash based approach to union:
  - Partition R and S using hash function *h*.
  - For each S partition, build in-memory hash table (using *h2*), scan corresponding R partition and add tuples to hash table while discarding duplicates
  - When done with partition, write out hashtable as (part of) result

Set Operations

- Set difference (R – S) similar to union
- Sorting-based approach
  - During merge pass, write out tuples in R after checking that do not appear in S
- Hashing-based approach
  - For each tuple of R, probe hashtable for partition of S and only write tuple (to output) if *not* found in hashtable.

Aggregate Operations (AVG, MIN, etc.)

- Without grouping:
  - In general, requires scanning the relation.
  - Keep track of some "running information"
    - SUM?
    - MAX/MIN?
    - AVG?

Aggregate Operations (AVG, MIN, etc.)

- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group.
    - Can improve upon this by combining sorting and aggregate computation (total cost = cost of sort in this case)
  - Hashing approach: build a hash table with entries <grouping-value, running-info> and update it as you scan the entire relation

Aggregate Operations (AVG, MIN, etc.)

- May be able to use indexes
  - Index-only scan
  - Retrieve records in sorted order instead of having to sort

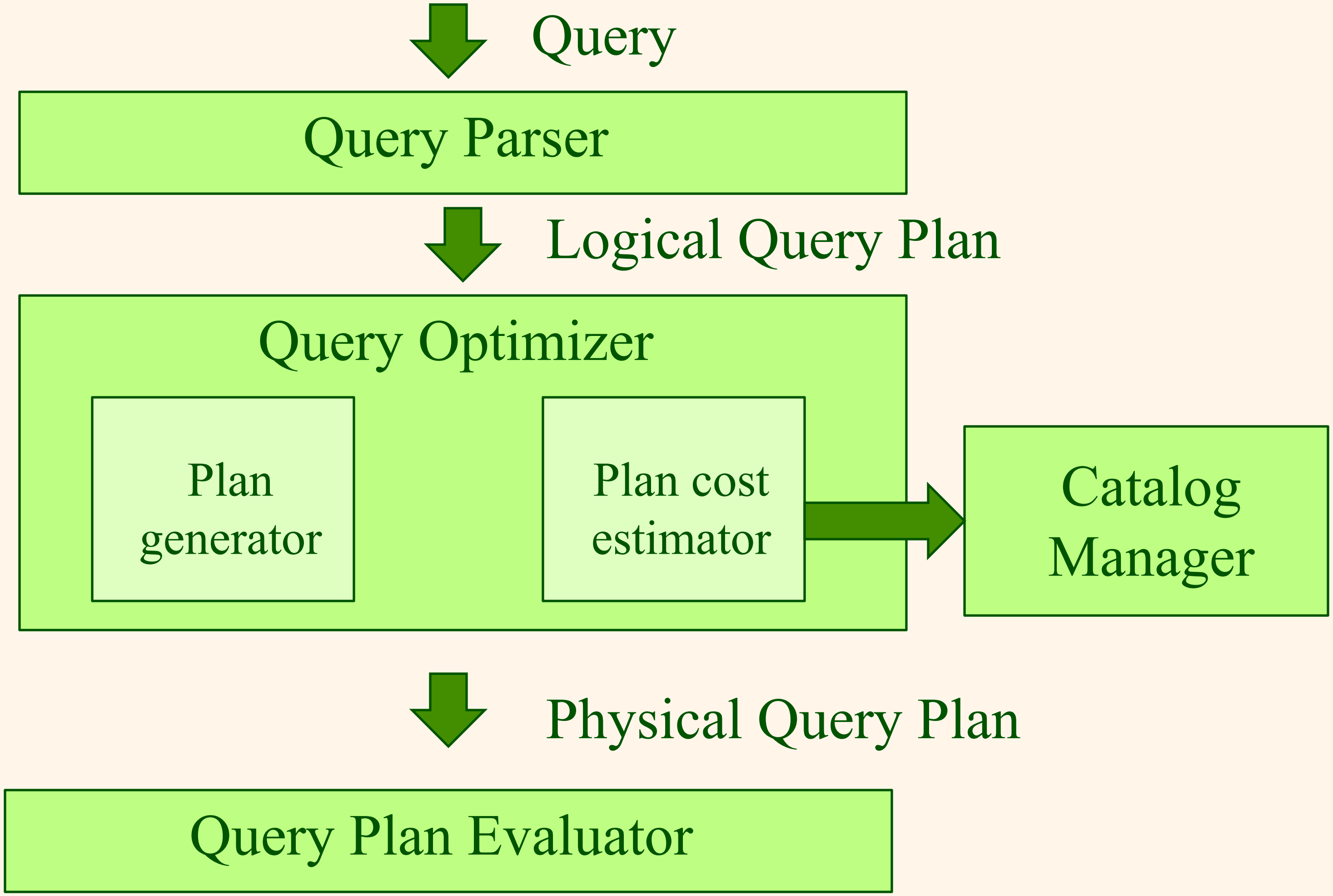
Summary so far

- Understand how to implement basic Relational Algebra operators
  - Select
  - Project
  - Join
  - Set operators
  - Aggregation/GROUP BY
- Understand that other implementations may be appropriate if setting/requirements are different

Putting it all together

- How to use these implementation algorithms to process your SQL queries efficiently?
- Query evaluation and optimization

Putting it all together



Parsing and decomposition

```
SELECT  S.sname,  S.age
FROM  Sailors S
WHERE  S.age =
          (SELECT  MAX(S2.age)
           FROM  Sailors S2);
```

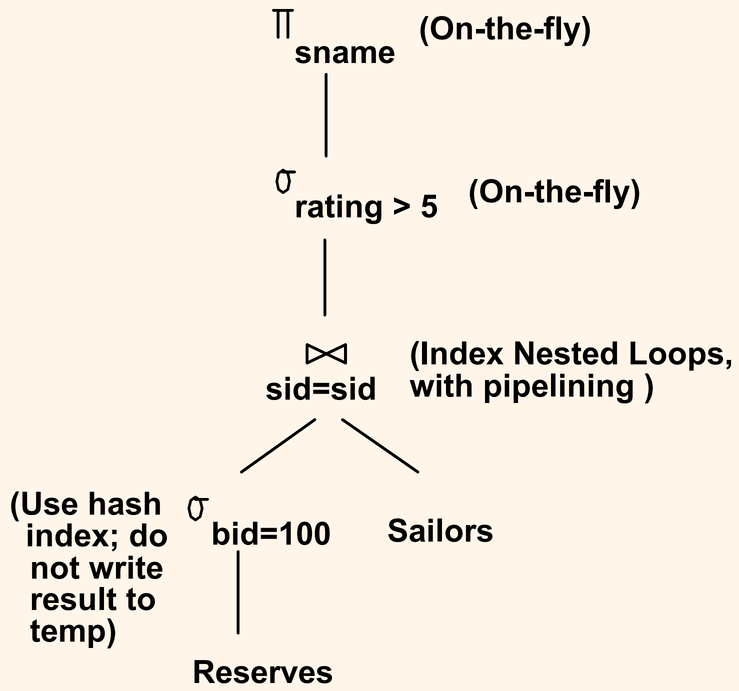
- This query will generate two **blocks**
- Blocks correspond to a single SELECT-FROM-WHERE clause
- Blocks optimized one at a time

Optimizing a block

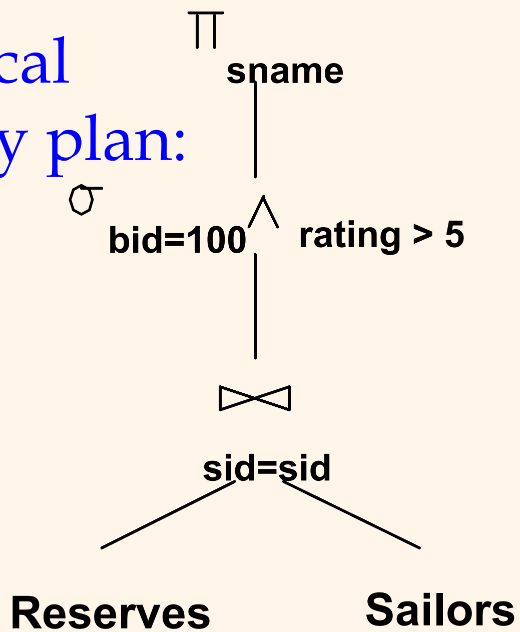
- A block is basically a Relational Algebra select-project-join ($\sigma \pi \bowtie$) expression
- With additional "operators"/annotations to handle features like
  - Aggregation
  - GROUP BY
  - ORDER BY
  - Etc
- Core of the optimizer's work: find the best **physical query plan** for the SPJ part

Query & logical and physical plans

SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5



Logical query plan:



- Physical query plan = RA tree annotated with info on access methods and operator implementation

The work of an optimizer

- Generate some different physical plans
  - Reorder operators (using our handy RA equivalences)
  - Experiment with different implementations for each operator
- For each plan, estimate the cost
- Choose the lowest-cost plan

The work of an optimizer

- Ideally: Want to find best plan. Practically: Avoid worst plans!
- Optimization can't take too long, otherwise might defeat the purpose (if takes longer to optimize than to run a "dumb" plan)

Two main questions

- How to generate (a good subset of) the possible plans?
- How to compute the cost of each plan?

- Let's start with the second question….
  - Basically it's about putting together the per-operator calculations we have been doing already

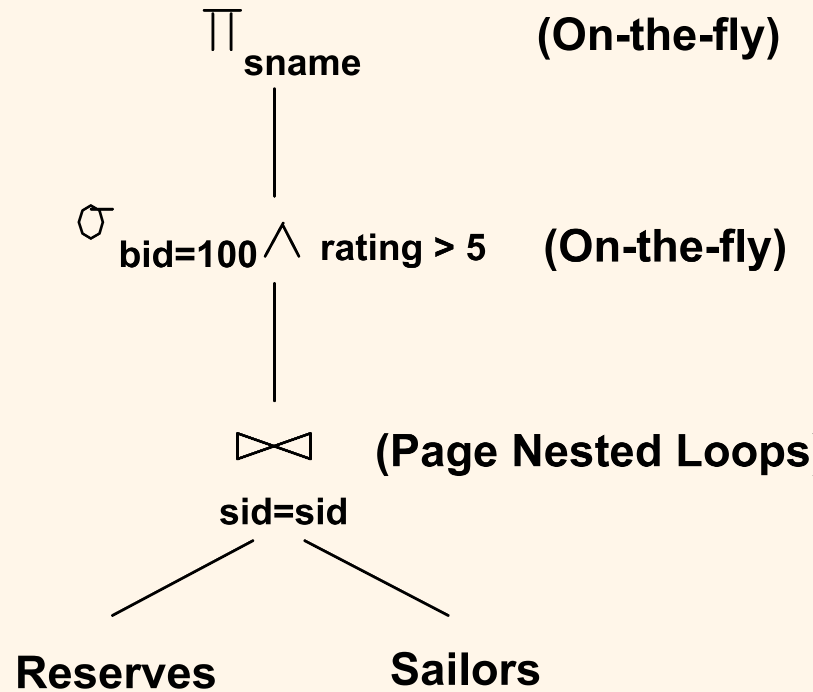
Let's look at some examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: date, *rname*: string)

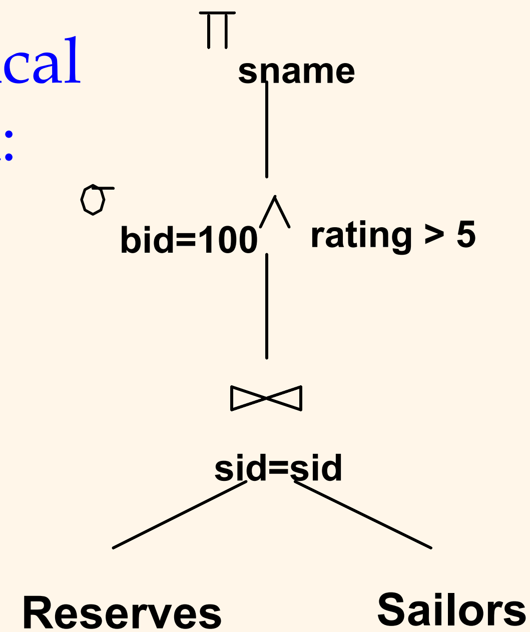
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

A first physical query plan

SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5



Logical plan:



- Cost: 1000+500*1000 = 501,000 page I/Os
- Convention: **left child** of join = **outer relation**

FAQ: what does "on the fly" mean?

- Just means that we can apply the operation while the tuple is already in memory
- So no extra I/O cost